Implementation:

Syscalls:

```
        void addTaskManager(TaskManager *tm);

    private:
        TaskManager* tm; //this is added to handle syscalls that need task manager.
    };
}

/*syscall numbers in enum.*/
enum SYSCALL{FORK = 2, WAITPID = 7, EXECVE = 59, PRINT = 4, GETPID = 39, EXIT = 1};


#endif
```

I added these inside the SyscallHandler class. The syscall handler enum is used to make system calls using inline assembly in the syscall functions in userspace.

Like: asm("int $0x80" :: "a" (SYSCALL::FORK));

I included a pointer to the task manager to call some task manager functions when an interrupt that needs those functions is occurred.

```
// system calls.

uint32_t sysFork(){
    asm("int $0x80" :: "a" (SYSCALL::FORK));
}

uint32_t sysWaitPid(uint32_t pid){
    asm("int $0x80" :: "a" (SYSCALL::WAITPID), "b" (pid));
}


void sysExecve(void entrypoint()){
    asm("int $0x80" : : "a" (SYSCALL::EXECVE), "b" ((uint32_t)entrypoint));
}

void sysPrintf(char* str)
{
    asm("int $0x80" : : "a" (SYSCALL::PRINT), "b" (str));
}

```

```
uint32_t sysGetPid()
{
    asm("int $0x80" :: "a" (SYSCALL::GETPID));
}

void sysExit()
{
    asm("int $0x80" ::  "a" (SYSCALL::EXIT));
}

```

I added these functions in the kernel.cpp file. (sysPrintf was already in there, I just changed the call input from 4 to SYSCALL::PRINT.)

Here, these functions cause 0x80 (syscall) interrupts by putting the desired call into eax register and some other arguments to other registers if necessary. These interrupts are handled by the SyscallHandler class, if the system call need to return a value, the eax register is set directly in the handler.

```
uint32_t SyscallHandler::HandleInterrupt(uint32_t esp)
{
    CPUState* cpu = (CPUState*)esp;

    switch(cpu->eax)
    {
        case SYSCALL::FORK:
            printf("syscall fork called\n");
            tm->Fork(cpu);
            break;

        case SYSCALL::WAITPID:
            printf("syscall waitpid called\n");
            cpu->eax = tm->WaitPid(cpu->ebx);
            esp = (uint32_t)tm->Schedule((CPUState*)esp); // c
            break;

        case SYSCALL::EXECVE:
            printf("syscall execve called\n");
            esp = (uint32_t) tm->ChangeCurrentTask(cpu->ebx);
            break;

        case SYSCALL::PRINT:
            printf((char*)cpu->ebx);
            break;

        case SYSCALL::GETPID:
            cpu->eax = tm->GetPid();
            break;

        case SYSCALL::EXIT:
            tm->Exit();
            esp = (uint32_t)tm->Schedule((CPUState*) esp);
            break;

        default:
            break;
    }

    return esp;
}
```

This is the HandleInterrupt function in the SyscallHandler class. I called the necessary functions in the taskmanager for each wanted system call (taken from cpu->eax), and then I put the return value to the return register (cpu->eax) if that system call has to return a value.

Exception here is the Fork call. Because the child and parent process after fork has to return different values, I set them in the Fork function in the taskmanager directly.

Wait and Exit syscalls call scheduler explicitly because those tasks won't be running after the call and we need to decide the next task.

TaskManager:

Task:

```
class Task
{
friend class TaskManager;

enum State {BLOCKED, READY, RUNNING, TERMINATED};

private:
    common::uint8_t stack[4096]; // 4 KiB

    CPUState* cpustate;

    State state = READY; //todo: set this after some checks?

    common::uint32_t pid;
    common::uint32_t p_pid;
    common::uint32_t waiting_child_pid;
public:
    Task(GlobalDescriptorTable *gdt, void entrypoint());
    Task();
    ~Task();
};
```

I added variables for state, pid, parent pid, the pid of the child that this task is waiting for. I added a default constructor explicitly because I hold the tasks in the TaskManager directly (not pointers to them.) I did this to make sure that the tasks are in the stack.

```
class TaskManager
{
private:
    Task tasks[256];
    int numTasks;
    int currentTask;
    int newPid;
    GlobalDescriptorTable* gdt;
    Task* TaskFromPid(common::uint32_t pid);

```

I added a pid counter named newPid to set the pid of a new task.

Task Manager Functions:

Schedule:

```cpp
CPUState* TaskManager::Schedule(CPUState* cpustate)
{
    printf("Schedular is called\n");
    // PrintTasks(tasks, numTasks);

    if(numTasks <= 0)
        return cpustate;

    if(currentTask >= 0)
        tasks[currentTask].cpustate = cpustate;


    // stop the current process.
    if((tasks[currentTask].state != Task::BLOCKED) && (tasks[currentTask].state != Task::TERMINATED))
        tasks[currentTask].state = Task::READY; // maybe some checks here.


    // (the quanta is the time between hardware timer interrupts)
    // Searches the next ready process.
    do{
        ++currentTask;

        if(currentTask >= numTasks)
            currentTask %= numTasks;

    }while(tasks[currentTask].state != Task::READY || tasks[currentTask].waiting_child_pid != -1);
    // I added the second check in while loop to pass the processes that are waiting for its child to finish.

    tasks[currentTask].state = Task::RUNNING;

    return tasks[currentTask].cpustate;
}
```

The schedule function first saves the current cpu state to the cpu state of the currently running task and make currently running task ready. I added some checks to make sure that scheduler won't unintentionally make a blocked or terminated process ready. After this, it searches for the next ready (and not waiting for a child) task in a loop. When it founds that task, it sets that tasks state to running and returns the cpustate of that task like older version.

```cpp
uint32_t TaskManager::GetPid(){
    return tasks[currentTask].pid;
}

uint32_t TaskManager::GetParrentPid(){
    return tasks[currentTask].p_pid;
}

uint32_t TaskManager::WaitPid(uint32_t pid){
    // todo: check this before in the syscall to preve
    if(TaskFromPid(pid)->state == Task::TERMINATED)
        return -1;

    tasks[currentTask].waiting_child_pid = pid;
    return pid;
}

void TaskManager::Exit(){
    Task* current = &tasks[currentTask];

    Task* parent = TaskFromPid(current->p_pid);

    if(current->p_pid != 0){
        Task* parent = TaskFromPid(current->p_pid);

        if(parent == nullptr){
            printf("Can't find parent in exit.\n");
        }

        if(parent->waiting_child_pid == current->pid)
            parent->waiting_child_pid = -1;
    }
    current->state = Task::TERMINATED;
}

Task* TaskManager::TaskFromPid(uint32_t pid){
    for(int i = 0; i < numTasks; ++i){
        Task* taskP = &tasks[i];
        if(taskP->pid == pid)
            return taskP;
    }
    return nullptr;
}
```

The getpid, getParrentPid functions are simply return the pid and p_pid.

WaitPid function sets the waitingPid of the current task to the pid from the syscall argument that is set in syscallhandler if that task is not terminated already.

The TaskFromPid returns the pointer to the task that its pid is pid in the argument.

Exit function first checks whether exited processes parent is waiting for exited process to finish, if so, it sets the waiting pid of the exited processes parent to -1 (I used -1 as that process is not waiting for a child to finish.) and then it sets the exited processes state as terminated.

Fork:

```cpp
void TaskManager::Fork(CPUState* cpuState){
    if(numTasks >= 256){
        cpuState->eax = -1;
        return;
    }
    Task * parent = &tasks[currentTask];
    Task * child = &tasks[numTasks];


    child->state = Task::READY;
    child->p_pid = parent->pid;
    child->pid = newPid++;
    child->waiting_child_pid = -1;

    for (int i = 0; i < sizeof(parent->stack); i++)
    {
        child->stack[i]=parent->stack[i];
    }

    uint32_t parentCpuStackOffset = (uint32_t)cpuState - (uint32_t) (parent->stack);
    child->cpustate = (CPUState*)(((uint32_t) (child->stack)) + parentCpuStackOffset);

    uint32_t spOffset = (uint32_t)(parent->stack) - (uint32_t) (cpuState);
    child->cpustate->esp = (uint32_t) (child->stack) + spOffset;

    child->cpustate->eax = 0;
    cpuState->eax = child->pid;
    numTasks++;
}
```

My fork function is almost as same as the shared code, except I fixed some bugs. Here I add a child process to the end of the tasks array and assign its state, parent pid, pid and waiting_child_pid. Then copy the stack. And calculate the cpu state and assign it also.

But after this the esp register of the child is same with the parent (that value is copied when we copy the stack) so we need to assign it to the correct value which is an offset from child stack. I calculated this offset by parent stack – current esp (since the stack pointer is incremented by decreasing it.) And I used the current cpuState as current esp because they are essentially same (in syscallhandler the current esp is set as cpustate).

Second change I did is, I used the return value register (eax) to set the fork return values of the child and parent correctly.

Helper Additions:

```c
void sleep(float seconds){
        for(uint32_t i = 0; i < (uint32_t)(999999.0 * seconds); ++i)
            for(uint32_t j = 0; j < 999; ++j);
}

// basic function to print decimal numbers.
// store the numbers while dividing by 10 and reverse string

void printfDec(int32_t decimal){
    int i = 0;
    bool isNegative = false;
    char str[16];

    if (decimal < 0) {
        isNegative = true;
        decimal = -decimal;
    }

    do {
        str[i++] = decimal % 10 + '0';
        decimal /= 10;
    } while (decimal != 0);

    if (isNegative) {
        str[i++] = '-';
    }

    int j = 0;
    char temp;
    for (j = 0; j < i / 2; j++) {
        temp = str[j];
        str[j] = str[i - j - 1];
        str[i - j - 1] = temp;
    }

    str[i] = '\0';
    printf(str);
}
```

I added a sleep function, it just counts in a nested for loop, I tuned the values for my computer so that argument 1 approximately counts for 1 second but these values may need to be changed if sleep takes longer or shorter time.

I added a simple function to print decimal values.

```c
    if(interrupt == hardwareInterruptOffset)
    {
        if(timerInterruptAmount++ >= SCH_SLW_DWN){
            esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
            timerInterruptAmount = 0;
        }
        else{
            sleep(0.0005f);
        }
    }
```

I added this check to slow down the scheduler.

1 / [SCH_SLW_DWN] hardware interrupt calls schedule function, on remaining interrupts, the kernel sleeps for very short amount of time.

In my code, value of SCH_SLW_DWN is defined as 99.

Collatz Sequence function:

```c
void collatzSequence() {
    int32_t r = random();

    if(r < 0)
        r *= -1;
    if(r == 0)
        r = 1;

    int n = r % 100;

    if (n <= 0) {
        sysPrintf("Input must be a positive integer.\n");
        return;
    }

    sysPrintf("Collatz sequence for ");
    sysPrintfDec(n);
    sysPrintf(" is: ");

    while (n != 1) {
        sysPrintfDec(n);
        sysPrintf(" ");
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    sysPrintf("1\n");
    sleep(0.5);

    sysExit();
}
```

Long Running Program:

```c
void long_running_program() {
    sysPrintf("Process id: ");
    sysPrintfDec(sysGetPid());
    sysPrintf("\n");


    int n = 99999;
    int result = 0;
    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {
            result += i * j;
        }
    }

    sleep(0.5);
    sysPrintf("result: ");
    sysPrintfDec(result);
    sysPrintf("\n");
    sysExit();
}
```

The programs are implemented as in the pdf, Collatz sequence program calculated the collatz sequence of a random number.

```c
int32_t random(){
    int32_t time;
    asm("rdtsc": "=A"(time));

    time *= time;
    time /= 100;
    return (int32_t)(time % 10000);
}
```

Random function is a very simple function that takes the time from rdtsc and returns it after some random calculations.

```c
void initialTask(){
    int32_t pids[6];

    for(int32_t i = 0; i < 6; i++) {
        if(i%2 == 0){
            pids[i] = sysFork();
            if(pids[i] == 0){
                sysPrintf("Forked and executing collatz..\n");
                sysPrintf("Pid is: ");
                sysPrintfDec(sysGetPid());
                sysPrintf("\n");
                sysExecve(collatzSequence);
                sleep(0.5);
            }
        }
        else{
            pids[i] = sysFork();
            if(pids[i] == 0){
                sysPrintf("Forked and executing long running program..\n");
                sysPrintf("Pid is: ");
                sysPrintfDec(sysGetPid());
                sysPrintf("\n");
                sysExecve(longRunningProgram);
                sleep(0.5);
            }
        }
    }

    for(int32_t i = 0; i < 6; ++i){
        sysWaitPid(pids[i]);
    }

    while(true){
        sysPrintf("Processes have finished. \n");
        sleep(0.5);
    }

}
```

I added this initial task that loads each program 3 times. After that, it waits for each of them, and then it prints “Processes have finished” in a while loop. My waitpid implementation takes a single process to wait. So in order to wait all of them, I called all of them in a loop. Since sysWaitPid return immediately if the task that we are trying to wait is done already, this didn’t cause any problems.

Tests:

Fork and GetPid:

I made a simple program that forks 2 times (second fork is called only in the child). Then all these processes print their process id using sysGetPid call and the return values from fork. The output is as expected:

```
Hello World! --- http://www.AlgorithMan.de
Hello
INTERRUPT FROM AMD am79c973
AMD am79c973 DATA SENT
AMD am79c973 INIT DONE
Schedular is called
I'm the process to test fork, I am going to fork and my child will fork too.
We will be printing our pids and forkReturn values.
syscall fork called
Hello, I'm the parent process, and my pid is: 1
 I'm going to print parent from now on.
Parent pid: 1, forkReturnParent: 2
Parent pid: 1, forkReturnParent: 2
Parent pid: 1, forkReturnParent: 2
Parent pid: 1, forkReturnParent: 2
Parent pid: 1, forkReturnParent: 2
Parent pid: 1, forkReturnParent: 2
Schedular is called
Hello, I'm the child process, and my pid is: 2
 I'm going to fork again and print child from now on.
syscall fork called
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
```

```
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
Schedular is called
Hello, I'm the grand child process, and my pid is: 3
 I'm going to print grand child from now on.
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Schedular is called
Parent pid: 1, forkReturnParent: 2
Parent pid: 1, forkReturnParent: 2
Parent pid: 1, forkReturnParent: 2
Parent pid: 1, forkReturnParent: 2
Parent pid: 1, forkReturnParent: 2
Parent pid: 1, forkReturnParent: 2
```

```
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
Child pid: 2, forkReturnParent: 0, forkReturnChild: 3
Schedular is called
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Grand Child pid: 3, forkReturnParent: 0, forkReturnChild: 0
Schedular is called
Parent pid: 1, forkReturnParent: 2
Parent pid: 1, forkReturnParent: 2
Parent pid: 1, forkReturnParent: 2
Parent pid: 1, forkReturnParent: 2
```

Execve:

In this test, I made a program like:

taskTestExecve:

    forkReturn = sysfork()

    if forkReturn == 0:

        execve(taskPrintExecve)

    while(true):

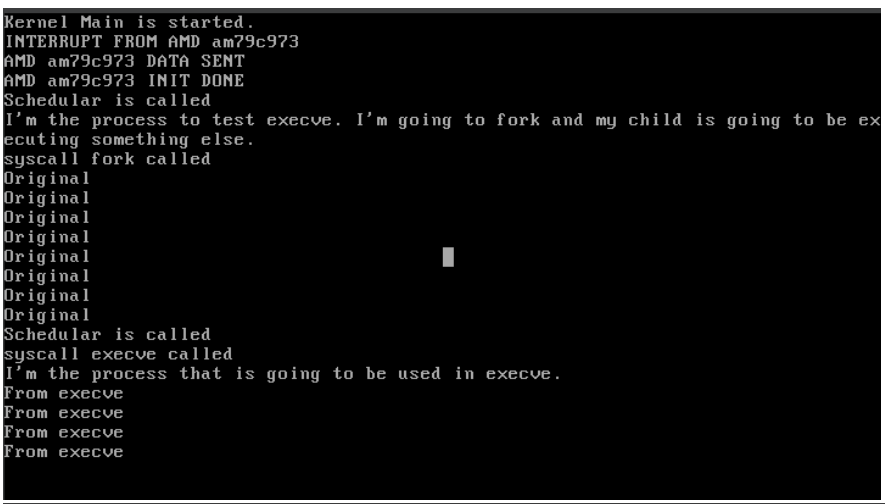
        print("original")

taskPrintExecve:

    while(true):

        print("from execve")

The output is as expected:

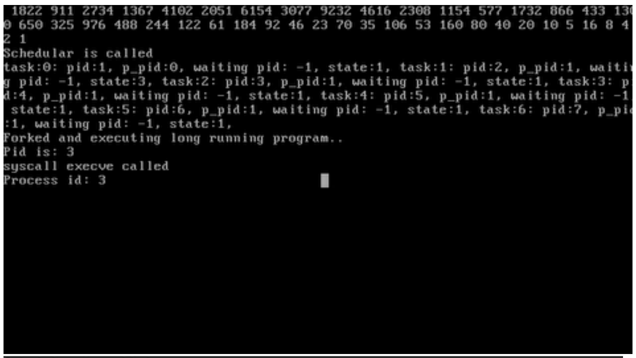


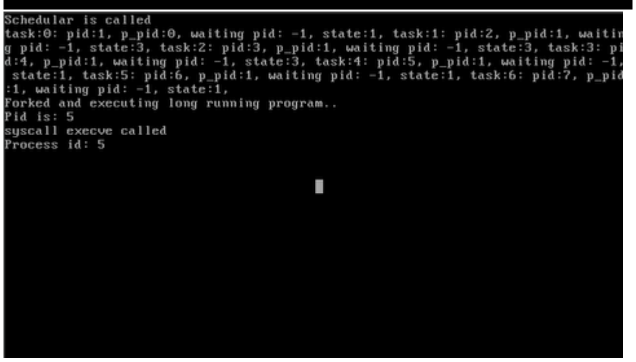
```
Kernel Main is started.
INTERRUPT FROM AMD am79c973
AMD am79c973 DATA SENT
AMD am79c973 INIT DONE
Schedular is called
I'm the process to test execve. I'm going to fork and my child is going to be ex
ecuting something else.
syscall fork called
Original
Original
Original
Original
Original
Original
Original
Original
Schedular is called
syscall execve called
I'm the process that is going to be used in execve.
From execve
From execve
From execve
From execve
```

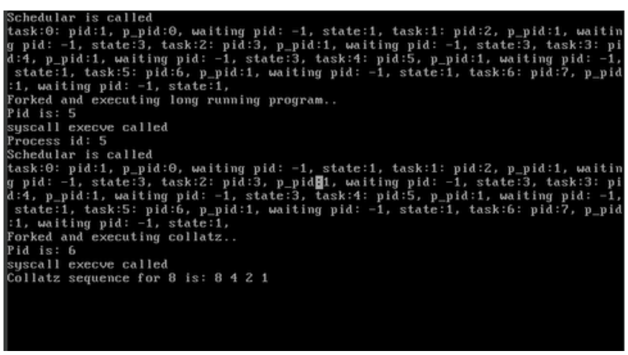
```
From execve
From execve
Schedular is called
Original
Original
Original
Original
Original
Original
Original
Schedular is called
From execve
From execve
From execve
From execve
From execve
From execve
From execve
Schedular is called
Original
Original
Original
```

My output:

With process table (with printing the process table, I can't show some parts of the output because even when I slowed the schedular and video record the output, when the printed task is big, it skips some of the output.):







```
:1, waiting pid: -1, state:1,
Forked and executing long running program..
Pid is: 7
syscall execve called
Process id: 7
result: 1884769809
Schedular is called
task:0: pid:1, p_pid:0, waiting pid: -1, state:1, task:1: pid:2, p_pid:1, waitin
g pid: -1, state:3, task:2: pid:3, p_pid:1, waiting pid: -1, state:3, task:3: pi
d:4, p_pid:1, waiting pid: -1, state:3, task:4: pid:5, p_pid:1, waiting pid: -1,
 state:1, task:5: pid:6, p_pid:1, waiting pid: -1, state:3, task:6: pid:7, p_pid
:1, waiting pid: -1, state:3,
syscall waitpid called
Schedular is called
task:0: pid:1, p_pid:0, waiting pid: -1, state:2, task:1: pid:2, p_pid:1, waitin
g pid: -1, state:3, task:2: pid:3, p_pid:1, waiting pid: -1, state:3, task:3: pi
d:4, p_pid:1, waiting pid: -1, state:3, task:4: pid:5, p_pid:1, waiting pid: -1,
 state:1, task:5: pid:6, p_pid:1, waiting pid: -1, state:3, task:6: pid:7, p_pid
:1, waiting pid: -1, state:3,
```

```
g pid: -1, state:3, task:2: pid:3, p_pid:1, waiting pid: -1, state:3, task:3: pi
d:4, p_pid:1, waiting pid: -1, state:3, task:4: pid:5, p_pid:1, waiting pid: -1,
 state:3, task:5: pid:6, p_pid:1, waiting pid: -1, state:3, task:6: pid:7, p_pid
:1, waiting pid: -1, state:3,
Processes have finished.
Schedular is called
task:0: pid:1, p_pid:0, waiting pid: -1, state:2, task:1: pid:2, p_pid:1, waitin
g pid: -1, state:3, task:2: pid:3, p_pid:1, waiting pid: -1, state:3, task:3: pi
d:4, p_pid:1, waiting pid: -1, state:3, task:4: pid:5, p_pid:1, waiting pid: -1,
 state:3, task:5: pid:6, p_pid:1, waiting pid: -1, state:3, task:6: pid:7, p_pid
:1, waiting pid: -1, state:3,
Processes have finished.
Processes have finished.
Processes have finished.
Processes have finished.
Processes have finished.
Processes have finished.
```

Without printing the process table (I put this output to show the results more clearly):

```
INTERRUPT FROM AMD am79c973
AMD am79c973 DATA SENT
AMD am79c973 INIT DONE
Schedular is called

syscall fork called
syscall fork called
syscall fork called
syscall fork called
syscall fork called
syscall fork called
syscall waitpid called
Schedular is called

Forked and executing collatz..
Pid is: 2
syscall execve called
Collatz sequence for 69 is: 69 208 104 52 26 13 40 20 10 5 16 8 4 2 1
Schedular is called

Forked and executing long running program..
Pid is: 3
syscall execve called
Process id: 3
```

```
result: 1884769809
Schedular is called

Forked and executing collatz..
Pid is: 4
syscall execve called
Collatz sequence for 56 is: 56 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2
1
Schedular is called

Forked and executing long running program..
Pid is: 5
syscall execve called
Process id: 5
Schedular is called

Forked and executing collatz..
Pid is: 6
syscall execve called
Collatz sequence for 69 is: 69 208 104 52 26 13 40 20 10 5 16 8 4 2 1
```

```
Process id: 7
result: 1884769809
Schedular is called

syscall waitpid called
Schedular is called

result: 1884769809
Schedular is called

syscall waitpid called
Schedular is called

syscall waitpid called
Schedular is called

syscall waitpid called
Schedular is called

syscall waitpid called
Schedular is called

Processes have finished.
Processes have finished.
```